Section 3

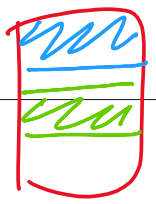
Wednesday, July 3, 2019 2:03 PM

# Section 3: Threads and Locks

CS162

July 3, 2019

## Contents



## 1 Vocabulary

- **Thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.
- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A `pthread_t` is usually just an alias for "`unsigned long int`".
- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally, this is implemented by calling the clone syscall.

```
/* On success, pthread_create() returns 0; on error, it returns an error
* number, and the contents of *thread are undefined. */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`.

```
/* On success, pthread_join() returns 0; on error, it returns an error number. */
int pthread_join(pthread_t thread, void **retval);
```

- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

```
/* On success, pthread_yield() returns 0; on error, it returns an error number. */
int pthread_yield(void);
```

- **Atomic operation** - An operation that appears to be indivisible to observers. Atomic operations must execute to completion or not at all.
- **Critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.
- **Race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.
- **Lock** - Synchronization variables that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads. See next_thread_to_run() in Pintos.
- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you're concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amended by implementing priority donation.

$i=0$

$0+25=25$

$i=25$

$i=0$

$0+40=40$

$i=40$

Value = 40

$\boxed{\cancel{10}}$ 50

$\boxed{0}$

- **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn't do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread's effective priority becomes the max of all donated priorities and its original priority.

## 2 Problems

### 2.1 Hello World

What does C print in the following code?

```
void* identify(void* arg) {
    pid_t pid = getpid();      123
    printf("My pid is %d\n", pid);
    return NULL;
}

int main() {                          123
    pthread_t thread;
    pthread_create(&thread, NULL, &identify, NULL);
    identify(NULL);
    return 0;
}
```

P C My pid is 123 | My pid is 123
C P My pid is 123 |

### 2.2 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    ~~pthread_yield();~~ → pthread_join(thread, NULL);
    printf("MAIN\n");
    return 0;
}
```

Helper | Main | Main
Main | Helper |

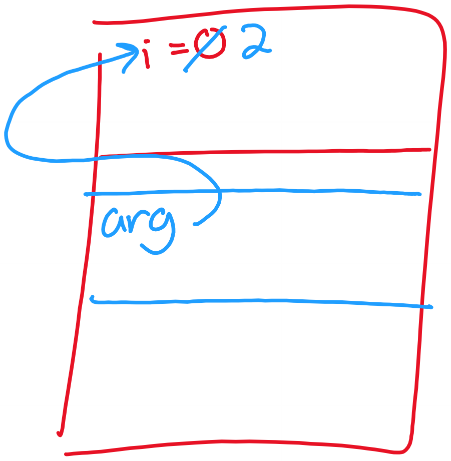
How can we modify the code above to always print out "HELPER" followed by "MAIN"?

### 2.3 Stack Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```



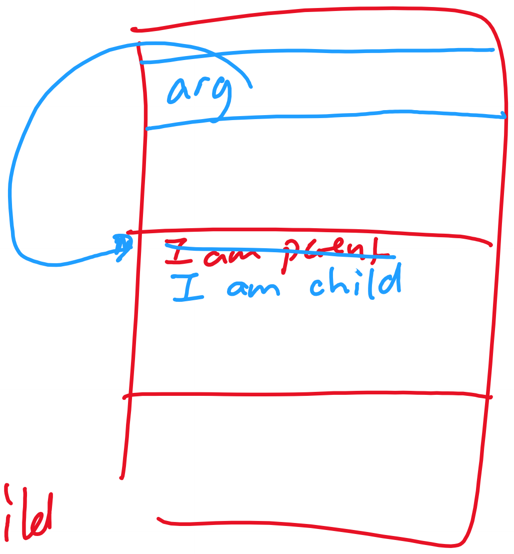
i is 2

### 2.4 Heap Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```



I am child

## 2.5 Threads and Processes

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *worker(void *arg) {
    int *data = (int *) arg;
    *data = *data + 1;
    printf("Data is %d\n", *data);
    return (void *) 42;
}

int data;
int main() {
    int status;
    data = 0;
    pthread_t thread;

    pid_t pid = fork();
    if (pid == 0) {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
    } else {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        wait(&status);
    }
    return 0;
}
```

How would you retrieve the return value of worker? (e.g. ”42”)

## 2.6 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
    assert (donor != recipient);

    if (donor->balance < amount) {
        printf("Insufficient funds.\n");
        return;
    }
    donor->balance -= amount;
    recipient->balance += amount;
}
```

acq lock →

rel lock →

1) read balance
2) sub amt
3) store back

| A | B |
|---|---|
| 5 | 5 |

A → B 5
A → B 5

| 0 | 10 |
|---|---|
| -5 | 15 |

Assume that there is some struct with a member `balance` that is `typedef`-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

| A | B |
|---|---|
| 5 | 5 |

A→B 5
B→A 5

5−5=0
A=0
5+5=10

5−5=0 ←
A=0+5=5 ←
B=10

| A | B |
|---|---|
| 5 | 10 |

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

## 2.7 All Threads Must Die

You have three Pintos threads with the associated priorities shown below. They each run the functions with their respective names.

Tyrion : 4
Ned: 5
Gandalf: ~~11~~ 3

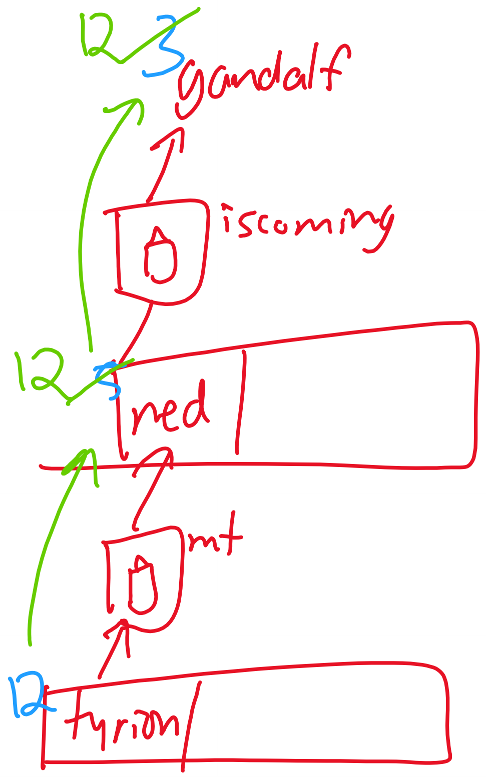
Assume upon execution that all threads are unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic. (Note: The following uses references to Pintos locks and data structures.)

```
struct list brace_yourself;   // pintos list. Assume it's already initialized and populated.
struct lock midterm;          // pintos lock. Already initialized.
struct lock is_coming;

void tyrion(){
 5 thread_set_priority(12);
 6 lock_acquire(&midterm);   block
   lock_release(&midterm);
   thread_exit();
}

void ned(){
 3 lock_acquire(&midterm);
 4 lock_acquire(&is_coming);  block
   list_remove(list_head(brace_yourself));
   lock_release(&midterm);
   lock_release(&is_coming);
   thread_exit();
}

void gandalf(){
 1 lock_acquire(&is_coming);
 2 thread_set_priority(3);
   while (thread_get_priority() < 11) {
       printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
       timer_sleep(20);
   }
   lock_release(&is_coming);
   thread_exit();
}
```



What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.